**Introduction**

When a company that is using DBT first starts experiencing problems with their data pipeline, the first thing they usually turn to is DBT tests.

DBT tests are a great first step for organizations that want to improve data quality and reliability, but they have their limitations. Some of this is inherent to testing: you only get information from the things you decide to test. Ideally, at some point, organizations shouldn’t just be testing their data pipelines: they should be moving on to data observability. In this blog post, we’ll cover 7 ways to go deeper with dbt tests, and when you should make the shift to a data observability solution.

### Understand the different types of dbt tests

There are two types of DBT tests:

**Data Tests** are specific queries that you run against a specific model. These live in the `tests` folder of your dbt project.

**Schema Tests** are generic tests that can be configured and applied to multiple models. Out of the box, dbt has four schema tests that can be immediately used: unique, not\_null, accepted\_values, and relationships. If you want to write your own custom schema tests, these are written as macros with the prefix test and stored in the `macros` folder.

If you have one model that you want to validate in a specific way, you can simply write a data test in the tests folder.

If you find yourself copy+pasting that query and configuring it for multiple models, then you may consider changing this from a data test over to a schema test so that you can write the test in one place and apply it to multiple models.

### Use plugins and packages

Dbt has a rich ecosystem of packages that integrate with it, including:

* [OpenLineage](https://openlineage.io/) - an open platform for collection and analysis of data lineage. It tracks metadata about datasets, jobs, and runs. OpenLineage consumes data produced by dbt runs.
* [dbt\_expectations](https://hub.getdbt.com/calogica/dbt_expectations/0.1.2/) - a dbt plugin that allows users to deploy GE-like tests in their data warehouse directly from dbt
* [Unit Tests for Macros](https://godatadriven.com/blog/dbts-missing-software-engineering-piece-unit-tests/)

### Schedule dbt tests for all dbt runs

Ideally, if you have a dbt job that runs daily, the associated dbt tests should run on the same schedule. To do this, you can [configure your dbt job to have multiple commands](https://docs.getdbt.com/blog/dbt-production-commands), including tests, running, freshness checks, and more. If any step fails you can get an email and a slack notification.

### Run DBT Tests on a model change/new PR

Whenever a PR is created in your database repo with a migration, you should use dbt test to verify that the changes won’t break your dbt models before merging the PR.

The best practice here is to use **data model blue-green deployment**. This means that your CI/CD tool should first do the `dbt run` and `dbt test` commands targeting a staging database environment OR a staging schema within the production environment. Then, if the tests pass, do `dbt run` against the production environment or the production schema.

To break this down:

**Blue-green data model testing**

1. Run `dbt run` in staging (either a separate staging schema or a separate staging database)
2. Run `dbt test` in staging
3. If all the tests pass, then run `dbt run` in prod

You should make sure that your staging environment is fully replicated so that the initial staging `dbt test` results are an accurate proxy for what will happen in production.

### 5. Run dbt test on your source models

To catch potential problems as early as possible, you should always run `dbt test` against your source models first:

1. Run dbt tests against the source
2. Run the dbt job
3. Test the resulting outputs

In other words:

dbt test --models source:[sources]

dbt run -m [models]

dbt test -m [models]

### 6. Distribute DBT Test Results

To make sure that you are actually aware when something your dbt transformations goes wrong, you should set up some interactive way to display and visualize your DBT test results.

There are a few tools, like [re\_data](https://github.com/re-data/re-data) and [elementary](https://github.com/elementary-data/elementary), that do this for you, or you can also spin up something custom: The data team at Snapcommerce, for example, [leveraged dbt artifacts](https://www.youtube.com/watch?v=LNY0K6mSEEI) and the query log to build reporting and alerts on dbt model/job runs and performance.

This may also be a good point to begin considering a more generic data observability tool, like Bigeye.

### 7. When appropriate, switch from testing to observability

While testing and observability have the same ultimate goal – detecting problems – they go about it in two different ways.

Testing is a human who has knowledge of the data expressing a condition that they know should be true. It is opinionated.

Observability, on the other hand, is unopinionated. It is purely asking, do we know the state of our data system at all times? Do we know how the state is changing? And will we know if the state changes in a way that is concerning?

A good analogy for observability is the dashboard on a vehicle. You can do without it, but most people would be pretty uncomfortable not knowing how fast they're going, or if the engine light is on. On the other hand, it doesn’t make sense to test the speed or the engine light setting at all times.

It’s becoming increasingly apparent that data engineers are going to some trouble to bridge the gap between dbt tests and a more traditional observability solution. Even so, though, this kind of hacked-together setup only gives you adequate observability into dbt – you remain ignorant of the other parts of your data stack. This is where tools like Bigeye, etc can help, because it looks at the final data at rest after everything has been transformed.